for all notes click here -github

(oops-cat2 solution)

Q1. Compare and Contrast error and exception.

Exceptions and errors both are subclasses of Throwable class.

The error indicates a problem that mainly occurs due to the lack of system resources and our application should not catch these types of problems. Some of the examples of errors are system crash error and out of memory error. Errors mostly occur at runtime that's they belong to an unchecked type.

Exceptions are the problems which can occur at runtime and compile time. It mainly occurs in the code written by the developers. Exceptions are divided into two categories such as checked exceptions and unchecked exceptions.

Errors	Exceptions
1. Impossible to recover from an error	1. Possible to recover from exceptions
2. Errors are of type 'unchecked'	2. Exceptions can be either 'checked' or 'unchecked'
3. Occur at runtime	3. Can occur at compile time or run time
4. Caused by the application running environment	4. Caused by the application itself

Q2. What is the need of inheritance in C++?

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

The idea of inheritance implements the **is a** relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on.

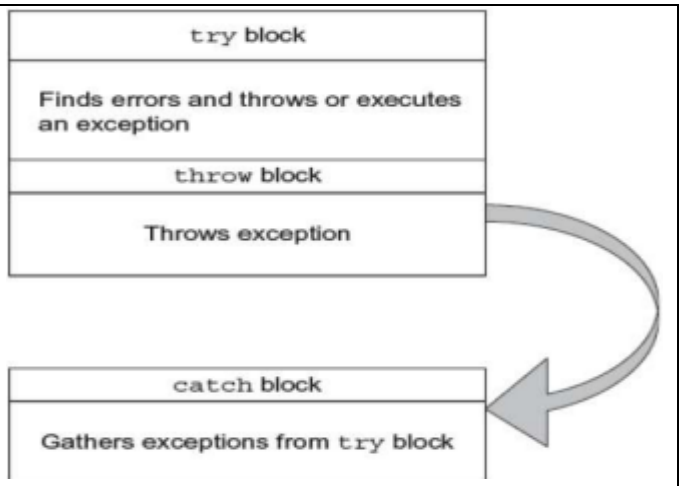
Q3. Draw a neat and clean diagram to show exception handling model in C++.

```
try {
        // protected code
        } catch( ExceptionName e1 ) {

        // catch block
        } catch( ExceptionName e2 ) {
        // catch block
        } catch( ExceptionName eN ) {
        // catch block
        }

        double division(int a, int b) {
        if( b == 0 ) {
        throw "Division by zero
        condition!";
        }
        return (a/b);
        }
```

diagram



Q4. What is Exception Handling in C++?

Exceptions are run-time anomalies or abnormal conditions that a program encounters during its execution.

There are two types of exceptions:

a)Synchronous,

b)Asynchronous(Ex:which are beyond the program's control, Disc failure etc). C++ provides following specialized keywords for this purpose.

try: represents a block of code that can throw an exception.

catch: represents a block of code that is executed when a particular exception is thrown.

throw: Used to throw an exception. Also used to list the exceptions that a function throws, but doesn't handle itself.

{

**(note - extra information- no need)

Following are main advantages of exception handling over traditional error handling.

1) Separation of Error Handling code from Normal Code: In traditional error handling codes, there are always if else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try catch blocks, the code for error handling becomes separate from the normal flow.

2) Functions/Methods can handle any exceptions they choose: A function can throw many exceptions, but may choose to handle some of them. The other exceptions which are thrown, but not caught can be handled by caller. If the caller chooses not to catch them, then the exceptions are handled by caller of the caller.

In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it)

3) Grouping of Error Types: In C++, both basic types and objects can be thrown as exception. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, categorize them according to types.

}

note **(imp.)

Exception handling in C++ consists of three keywords:
try, throw and catch:

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The throw keyword throws an exception when a problem is detected, which lets us create a custom error.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

The try and catch keywords come in pairs:

We use the try block to test some code: If the age variable is less than 18, we will throw an exception, and handle it in our catch block.

In the catch block, we catch the error and do something about it. The catch statement takes a parameter: in our example we use an int variable (myNum) (because we are throwing an exception of int type in the try block (age)), to output the value of age.

If no error occurs (e.g. if age is 20 instead of 15, meaning it will be be greater than 18), the catch block is skipped:

example

```
#include <iostream>
using namespace std;
int main(){
   try  {
     throw 10;
}
   catch (char *excp)  {
      cout << "Caught " << excp;
   }
   catch (...)  {
      cout << "Default Exception\n";
   }
   return 0;
}
```

Default Exception

Q5. Define dynamic binding.

The general meaning of binding is linking something to a thing. Here linking of objects is done. In a programming sense, we can describe binding as linking function definition with the function call.

So the term dynamic binding means to select a particular function to run until the runtime. Based on the type of object, the respective function will be called as dynamic binding .dynamic binding provides flexibility, it avoids the problem of static binding as it happened at compile time and thus linked the function call with the function definition.

A function declared in the base class and overridden(redefined) in the child class is called virtual function. When we refer derived class object using a pointer or reference to the base, we can call a virtual function for that object and execute the derived class's version of the function.

ex.

input	output

#include <iostream>
using namespace std;
class A {
public:
void final_print() {
display();
}
virtual void display(){
cout<< "Printing from the base class" <<endl;
}
};
class B : public A {
public:
virtual void display() {
cout<< "Printing from the derived class" <<endl;
}
};
int main(){
A obj1; // Creating A's pbject
obj1.final_print(); // Calling final_print
B obj2; // calling b
obj2.final_print();
return 0;
}
	Printing from the base class
	Printing from the derived class

Q6. What is an abstract class?

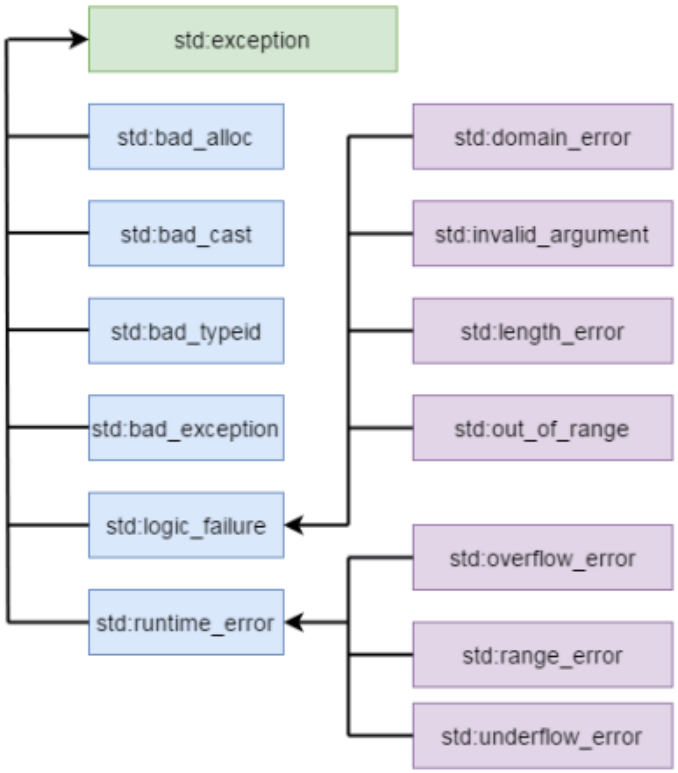
An *abstract class* is a class that is designed to be specifically used as a base class. An abstract class contains at least one *pure virtual function*. You declare a pure virtual function by using a *pure specifier* (= 0) in the declaration of a virtual member function in the class declaration.

The following is an example of an abstract class:

```
class AB {

public:
  virtual void f() = 0;
};
```

Q7. List out five common examples of exceptions.



Exception	Description
std::exception	It is an exception and parent class of all standard C++ exceptions.
std::logic_failure	It is an exception that can be detected by reading a code.
std::runtime_error	It is an exception that cannot be detected by reading a code.
std::bad_exception	It is used to handle the unexpected exceptions in a c++ program.
std::bad_cast	This exception is generally be thrown by **dynamic_cast.**
std::bad_typeid	This exception is generally be thrown by **typeid.**
std::bad_alloc	This exception is generally be thrown by **new.**

Q8. How overriding is different from the overloading?

S.NO	Method Overloading	Method Overriding
1.	Method overloading is a compile-time polymorphism.	Method overriding is a run-time polymorphism.
2.	It helps to increase the readability of the program.	It is used to grant the specific implementation of the method which is already provided by its parent class or superclass.
3.	It occurs within the class.	It is performed in two classes with inheritance relationships.
4.	Method overloading may or may not require inheritance.	Method overriding always needs inheritance.
5.	In method overloading, methods must have the same name and different signatures.	In method overriding, methods must have the same name and same signature.
6.	In method overloading, the return type can or can not be the same, but we just have to change the parameter.	In method overriding, the return type must be the same or co-variant.

Q9. Illustrate unexpected () function with an example.

When a function with an exception specification throws an exception that is not listed in its exception specification, the C++ run time does the following:

1. The `unexpected()` function is called.
2. The `unexpected()` function calls the function pointed to by `unexpected_handler`. By default, `unexpected_handler` points to the function `terminate()`.

You can replace the default value of `unexpected_handler` with the function `set_unexpected()`.

Although `unexpected()` cannot return, it may throw (or rethrow) an exception. Suppose the exception specification of a function `f()` has been violated. If `unexpected()` throws an exception allowed by the exception specification of `f()`, then the C++ run time will search for another handler at the call of `f()`. The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct E {
  const char* message;
  E(const char* arg) : message(arg) { }
};

void my_unexpected() {
  cout << "Call to my_unexpected" <<
endl;
  throw E("Exception thrown from
my_unexpected");
}

void f() throw(E) {
  cout << "In function f(), throw const
char* object" << endl;
  throw("Exception, type const char*,
thrown from f()");
}
int main() {
  set_unexpected(my_unexpected);
  try {
    f();
  }
  catch (E& e) {
    cout << "Exception in main(): " <<
e.message << endl;
  }
}
```

```
In function f(), throw const char*
object
Call to my_unexpected
Exception in main(): Exception
thrown from my_unexpected
```

Q10. What is a user defined exception? What are the advantages of using exception handling mechanism in a program? When do we use multiple ***catch*** handlers?

There maybe situations where you want to generate some user/program specific exceptions which are not pre-defined in C++. In such cases C++ provided us with the mechanism to create our own exceptions by **inheriting** the **exception** class in C++ and **overriding** its functionality according to our needs

the benefits of exception handling are as follows,

(a) Exception handling can control run tune errors that occur in the program.

(b) It can avoid abnormal termination of the program and also shows the behavior of program to users.

(c) It can provide a facility to handle exceptions, throws message regarding exception and completes the execution of program by catching the exception

(d) It can separate the error handling code and normal code by using try-catch block.

(e) It can produce the normal execution flow for a program.

(f) A failed constructor can also be handled by throwing an exception. The code for try, catch and throw blocks must be written in the constructor itself.

Multiple catch blocks are used when we have to catch a specific type of exception out of many possible type of exceptions i.e. an exception of type *char* or *int* or *short* or *long* etc.

user defined exception	output

#include <iostream>
#include <exception>
using namespace std;
class MyException : public exception {
public:
char * what () {
return "C++ Exception";
}
};
int main() {
try {
throw MyException();
}catch(MyException e) {
cout << "MyException caught" <<endl;
cout << e.what() <<endl;
} catch(exception e) {
//Other errors
}
return 0;
}
	MyException caught
	C++ Exception

multiple ***catch*** handlers	output

#include<iostream>
using namespace std;
int main(){
int a=10, b=0, c;
try{
//if a is divided by b(which has a value 0);
if(b==0)
throw(c);
else
c=a/b; }
catch(char c){
cout<<"Caught exception : char type ";
}
catch(int i){
cout<<"Caught exception : int type ";
}
catch(short s){
cout<<"Caught exception : short type ";
}
cout<<"\n Hello";}
	Caught exception : int type
	Hello

Q11. Demonstrate single inheritance with suitable example.

Single Inheritance: In single inheritance, a class is allowed to inherit from only one class. i.e. one subclass is inherited by one base class only

```
#include<iostream>
using namespace std;
class Vehicle {
  public:
    Vehicle(){
      cout << "This is a Vehicle\n";
    }
};
class Car : public Vehicle {
};
int main(){
    Car obj;
    return 0;
}
```

```
This is a Vehicle
```

Q12. Write down a detailed C++ program to demonstrate the use of try, catch, throw.

```
#include <iostream>
using namespace std;
int main(){
   int x = -1;
   cout << "Before try \n";
   try {
      cout << "Inside try \n";
      if (x < 0){
         throw x;
         cout << "After throw (Never executed) \n";
      }
   }
   catch (int x ) {
      cout << "Exception Caught \n";
   }
   cout << "After catch (Will be executed) \n";
   return 0;
}
```

```
Before try
Inside try
Exception Caught
After catch (Will be executed)
```

Q13. Explain the use of terminate ( ) function in C++ with an example.

- By default, the *terminate handler* calls abort. But this behavior can be redefined by calling `set_terminate`.

  This function is automatically called when no `catch` handler can be found for a thrown exception, or for some other exceptional circumstance that makes impossible to continue the exception handling process.

  This function is provided so that the *terminate handler* can be explicitly called by a program that needs to abnormally terminate, and works even if `set_terminate` has not been used to set a custom *terminate handler* (calling `abort` in this case).
- **No-throw guarantee:** this function never throws exceptions.

```
// terminate example

#include <iostream>      // std::cout, std::cerr

#include <exception>     // std::exception, std::terminate

int main (void) {

 char* p;

 std::cout << "Attempting to allocate 1 GiB...";

 try {

  p = new char [1024*1024*1024];

 }

 catch (std::exception& e) {

  std::cerr << "ERROR: could not allocate storage\n";

  std::terminate();

 }

 std::cout << "Ok\n";

 delete[] p;

 return 0;

}
```

```
Attempting to allocate 1 GiB...Ok
```

Q14. Demonstrate multiple inheritance with suitable example.

Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes

```
#include<iostream>
using namespace std;
class A{
public:
  A()  { cout << "A's constructor called" << endl; }
};
class B{
public:
  B()  { cout << "B's constructor called" << endl; }
};
class C: public B, public A{
public:
  C()  { cout << "C's constructor called" << endl; }
};
int main(){
   C c;
   return 0;
}
```

```
B's constructor called
A's constructor called
C's constructor called
```

Q15. What do you mean by Rethrowing an exception? Describe which type of catch block is used to catch all types of exceptions in C++ and why?

If you want to rethrow an exception from within an exception handler, you can do so by calling throw by itself, with no exception.

This causes the current exception to be passed on to an outer try/catch sequence.

An exception can be rethrown only from within a catch block or from any function called from within that block.

When you rethrow an exception, it will not be *recaught* by the same catch statement.

It will propagate to the immediately enclosing try/catch sequence.

The following program illustrates rethrowing an exception.

It rethrows a char * exception

```
#include <iostream>
using namespace std;
void Xhandler(){
   try{
      throw "hello"; // throw a char *
   }
   catch (char *){ // catch a char *
      cout << "Caught char * inside Xhandler\n";
      throw; // rethrow char * out of function
   }
}
int main(){
   cout << "start\n";
   try{
      Xhandler();
   }
   catch (char *){
      cout << "Caught char * inside main\n";
   }
   cout << "end";
   return 0;
}
```

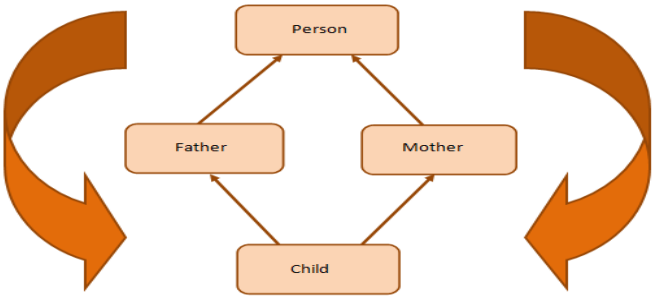
```
start

terminate called after throwing an instance of 'char const*'

Aborted
```

Q16. What is the diamond problem in C++? and explain how to fix it?

The Diamond Problem occurs when a child class inherits from two parent classes who both share a common grandparent class



As shown in the figure, class Child inherits the traits of class Person twice—once from Father and again from Mother. This gives rise to ambiguity since the compiler fails to understand which way to go.

This scenario gives rise to a diamond-shaped inheritance graph and is famously called "The Diamond Problem."

The solution to the diamond problem is to use the virtual keyword. We make the two parent classes (who inherit from the same grandparent class) into virtual classes in order to avoid two copies of the grandparent class in the child class

```
#include<iostream>
using namespace std;
class Person { //class Person
public:
   Person() { cout << "Person::Person() called" << endl; }
   Person(int x) { cout << "Person::Person(int) called" << endl; }
};
class Father : virtual public Person {
public:
   Father(int x):Person(x)   {
     cout << "Father::Father(int) called" << endl;
   }
};
class Mother : virtual public Person {
public:
   Mother(int x):Person(x) {
      cout << "Mother::Mother(int) called" << endl;
   }
};
class Child : public Father, public Mother  {
public:
   Child(int x):Mother(x), Father(x) {
      cout << "Child::Child(int) called" << endl;
   }
};
int main()  {
   Child child(30);}
```

```
Person::Person() called
Father::Father(int) called
Mother::Mother(int) called
Child::Child(int) called
```

Q17. When you will create class template? Write the syntax for creating class templates?

we can use class templates to create a single class to work with different data types.

Class templates come in handy as they can make our code shorter and more manageable

Syntax	

template <class T>
class className {
private:
T var;
... .
public:
T functionName(T arg);
... .
};	

example
#include <iostream>
using namespace std;
template <class T>
class Number {
private:
T num;
public:
Number(T n) : num(n) {}
T getNum() {
return num;
}
};
int main() {
Number<int> numberInt(7);
Number<double> numberDouble(7.7);
cout << "int Number = " << numberInt.getNum() << endl;
cout << "double Number = " << numberDouble.getNum() << endl;
return 0;
}
	int Number = 7
	double Number = 7.7

Q18. Discuss the role of access specifiers in an inheritance and show their visibility when they are inherited as public, private and protected.

public members can be accessed by anybody**. Private** members can only be accessed by member functions of the same class or friends. This means derived classes can not access private members of the base class directly!

The **protected** access specifier allows the class the member belongs to, friends, and derived classes to access the member. However, protected members are not accessible from outside the class.

Public inheritance

Access specifier in base class	Access specifier when inherited publicly
Public	Public
Protected	Protected
Private	Inaccessible

Protected inheritance

Access specifier in base class	Access specifier when inherited protectedly
Public	Protected
Protected	Protected
Private	Inaccessible

Private inheritance

Access specifier in base class	Access specifier when inherited privately
Public	Private
Protected	Private
Private	Inaccessible

Syntax for all access specifier

```
// Inherit from Base publicly
class Pub: public Base
{
};

// Inherit from Base protectedly
class Pro: protected Base
{
};

// Inherit from Base privately
class Pri: private Base
{
};

class Def: Base // Defaults to private inheritance
{
};
```

Q19. What is the difference between pure virtual functions and virtual functions?

Virtual function	Pure virtual function
A virtual function is a member function of base class which can be redefined by derived class.	A pure virtual function is a member function of base class whose only declaration is provided in base class and should be defined in derived class otherwise derived class also becomes abstract.
Classes having virtual functions are not abstract.	Base class containing pure virtual function becomes abstract.

Syntax:
`virtual<func_type><func_name>()`
`{`
`// code`
`}`
	Syntax:
	`virtual<func_type><func_name>()`
	`= 0;`

Definition is given in base class.	No definition is given in base class.
Base class having virtual function can be instantiated i.e. its object can be made.	Base class having pure virtual function becomes abstract i.e. it cannot be instantiated.
If derived class do not redefine virtual function of base class, then it does not affect compilation.	If derived class do not redefine virtual function of base class, then no compilation error but derived class also becomes abstract just like the base class.
All derived class may or may not redefine virtual function of base class.	All derived class must redefine pure virtual function of base class otherwise derived class also becomes abstract just like base class.

Q20. How overriding is different from the overloading?

S.NO	Method Overloading	Method Overriding
1.	Method overloading is a compile-time polymorphism.	Method overriding is a run-time polymorphism.
2.	It helps to increase the readability of the program.	It is used to grant the specific implementation of the method which is already provided by its parent class or superclass.
3.	It occurs within the class.	It is performed in two classes with inheritance relationships.
4.	Method overloading may or may not require inheritance.	Method overriding always needs inheritance.
5.	In method overloading, methods must have the same name and different signatures.	In method overriding, methods must have the same name and same signature.
6.	In method overloading, the return type can or can not be the same, but we just have to change the parameter.	In method overriding, the return type must be the same or co-variant.

```
#include <iostream>
using namespace std;
void print(int i) {
  cout << " Here is int " << i << endl;
}
void print(double  f) {
  cout << " Here is float " << f << endl;
}
void print(char const *c) {
  cout << " Here is char* " << c << endl;
}
int main() {
  print(10);
  print(10.10);
  print("ten");
  return 0;
}
```

Output

```
Here is int 10
Here is float 10.1
 Here is char* ten
```

```
#include<bits/stdc++.h>
using namespace std;
class a{
public:
     virtual void display(){ cout << "hello\n"; }
     void show(){cout<<"hello\n";}
};
class b:public a{
public:
      void display(){ cout << "bye\n";}
};
int main(){
    b obj1;
    a &ptr=obj1;
    ptr.display();
    obj1.show();
    return 0;
}
```

Output

```
bye
hello
```

brief note

Function Overloading (achieved at compile time)	Function Overriding (achieved at run time)
It provides multiple definitions of the function by changing signature i.e changing number of parameters, change datatype of parameters, return type doesn't play anyrole. It can be done in base as well as derived class	It is the redefinition of base class function in its derived class with same signature i.e return type and parameters. It can only be done in derived class.